Indian Hill C Style and Coding Standards
as amended for U of T Zoology UNIX†

L.W. Cannon
R.A. Elliott
L.W. Kirchhoff
J.H. Miller
J.M. Milner
R.W. Mitze
E.P. Schan
N.O. Whittington

Bell Labs

Henry Spencer

Zoology Computer Systems
University of Toronto

ABSTRACT

This document is an annotated (by the last author) version of the original paper of the same title. It describes a set of coding standards and recommendations which are local standards for officially-supported UNIX programs. The scope is coding style, not functional organization.

April 21, 1990

† UNIX is a trademark of Bell Laboratories.

Indian Hill C Style and Coding Standards as amended for U of T Zoology UNIX†

L.W. Cannon
R.A. Elliott
L.W. Kirchhoff
J.H. Miller
J.M. Milner
R.W. Mitze
E.P. Schan
N.O. Whittington

Bell Labs

Henry Spencer

Zoology Computer Systems
University of Toronto

1. Introduction

This document is a result of a committee formed at Indian Hill to establish a common set of coding standards and recommendations for the Indian Hill community. The scope of this work is the coding style, not the functional organization of programs. The standards in this document are not specific to ESS programming only[1]. We have tried to combine previous work [1,6] on C style into a uniform set of standards that should be appropriate for any project using C[2].

2. File Organization

A file consists of various sections that should be separated by several blank lines. Although there is no maximum length requirement for source files, files with more than about 1500 lines are cumbersome to deal with. The editor may not have enough temp space to edit the file, compilations will go slower, etc. Since most of us use 300 baud terminals, entire rows of asterisks, for example, should be discouraged[3]. Also lines longer than 80 columns are not handled well by all terminals and should be avoided if possible[4].

The suggested order of sections for a file is as follows:

1. Any header file includes should be the first thing in the file.
2. Immediately after the includes[5] should be a prologue that tells what is in that file. A description of the purpose of the objects in the files (whether they be functions, external data declarations or definitions, or something else) is more useful than a list of the object names.

---

† UNIX is a trademark of Bell Laboratories.

1. In fact, they're pretty good general standards. "To be clear is professional; not to be clear is unprofessional." — Sir Ernest Gowers. This document is presented unadulterated; U of T variations, comments, exceptions, etc. are presented in footnotes.
2. Of necessity, these standards cannot cover all situations. Experience and informed judgement count for much. Inexperienced programmers who encounter unusual situations should consult 1) code written by experienced C programmers following these rules, or 2) experienced C programmers.
3. This is not a problem at U of T, or most other sensible places, but rows of asterisks are still annoying.
4. Excessively long lines which result from deep indenting are often a symptom of poorly-organized code.
5. A common variation, in both Bell code and ours, is to reverse the order of sections 1 and 2. This is an acceptable practice.

3. Any typedefs and defines that apply to the file as a whole are next.
4. Next come the global (external) data declarations. If a set of defines applies to a particular piece of global data (such as a flags word), the defines should be immediately after the data declaration[6].
5. The functions come last[7].

2.1. File Naming Conventions

UNIX requires certain suffix conventions for names of files to be processed by the *cc* command [5][8]. The following suffixes are required:

- C source file names must end in *.c*
- Assembler source file names must end in *.s*

In addition the following conventions are universally followed:

- Relocatable object file names end in *.o*
- Include header file names end in *.h*[9] or *.d*
- Ldp[10] specification file names end in *.b*
- Yacc source file names end in *.y*
- Lex source file names end in *.l*

3. Header Files

Header files are files that are included in other files prior to compilation by the C preprocessor. Some are defined at the system level like *stdio.h* which must be included by any program using the standard I/O library. Header files are also used to contain data declarations and defines that are needed by more than one program[11]. Header files should be functionally organized, i.e., declarations for separate subsystems should be in separate header files. Also, if a set of declarations is likely to change when code is ported from one machine to another, those declarations should be in a separate header file.

Header files should not be nested. Some objects like typedefs and initialized data definitions cannot be seen twice by the compiler in one compilation. On non-UNIX systems this is also true of uninitialized declarations without the *extern* keyword[12]. This can happen if include files are nested and will cause the compilation to fail.

4. External Declarations

External declarations should begin in column 1. Each declaration should be on a separate line. A comment describing the role of the object being declared should be included, with the exception that a list of defined constants do not need comments if the constant names are sufficient documentation. The

---

6. Such defines should be indented to put the *defines* one level deeper than the first keyword of the declaration to which they apply.

7. They should be in some sort of meaningful order. Top-down is generally better than bottom-up, and a ''breadth-first'' approach (functions on a similar level of abstraction together) is preferred over depth-first (functions defined as soon as possible after their calls). Considerable judgement is called for here. If defining large numbers of essentially-independent utility functions, consider alphabetical order.

8. In addition to the suffix conventions given here, it is conventional to use 'Makefile' (not 'makefile') for the control file for *make* and 'README' for a summary of the contents of a directory or directory tree.

9. Preferred. An alternate convention that may be preferable in multi-language environments is to use the same suffix as an ordinary source file but with two periods instead of one (e.g. ''foo..c'').

10. No idea what this is.

11. Don't use absolute pathnames for header files. Use the *<name>* construction for getting them from a standard place, or define them relative to the current directory. The –I option of the C compiler is the best way to handle extensive private libraries of header files; it permits reorganizing the directory structure without having to alter source files.

12. It should be noted that declaring variables in a header file is often a poor idea. Frequently it is a symptom of poor partitioning of code between files.

comments should be tabbed so that they line up underneath each other[13]. Use the tab character (CTRL I if your terminal doesn't have a separate key) rather than blanks. For structure and union template declarations, each element should be alone on a line with a comment describing it. The opening brace ({) should be on the same line as the structure tag, and the closing brace should be alone on a line in column 1, i.e.

```
struct boat {
	int wllength;		/* water line length in feet */
	int type;	/* see below */
	long sarea;	/* sail area in square feet */
};
/*
 * defines for boat.type[14]
 */
#define	KETCH	1
#define	YAWL	2
#define	SLOOP	3
#define	SQRIG	4
#define	MOTOR	5
```

If an external variable is initialized[15] the equal sign should not be omitted[16].

```
	int x = 1;
	char *msg = "message";
	struct boat winner = {
		40,	/* water line length */
		YAWL,
		600	/* sail area */
	};
```

17

5. Comments

Comments that describe data structures, algorithms, etc., should be in block comment form with the opening /* in column one, a * in column 2 before each line of comment text[18], and the closing */ in columns 2-3.

---

13. So should the constant names and their defined values.

14. These defines are better put right after the declaration of *type*, within the *struct* declaration, with enough tabs after # to indent *define* one level more than the structure member declarations.

15. Any variable whose initial value is important should be *explicitly* initialized, or at the very least should be commented to indicate that C's default initialization to 0 is being relied on.

16. The empty initializer, "{}", should never be used. Structure initializations should be fully parenthesized with braces. Constants used to initialize longs should be explicitly long.

17. In any file which is part of a larger whole rather than a self-contained program, maximum use should be made of the *static* keyword to make functions and variables local to single files. Variables in particular should be accessible from other files only when there is a clear need that cannot be filled in another way. Such usages should be commented to make it clear that another file's variables are being used; the comment should name the other file.

18. Some automated program-analysis packages use a different character in this position as a marker for lines with specific items of information. In particular, a line with a '-' here in a comment preceding a function is sometimes assumed to be a one-line summary of the function's purpose.

```
/*
 *      Here is a block comment.
 *      The comment text should be tabbed over[19]
 *      and the opening /* and closing star-slash
 *      should be alone on a line.
 */
```

Note that *grep* ^.* will catch all block comments in the file. In some cases, block comments inside a function are appropriate, and they should be tabbed over to the same tab setting as the code that they describe. Short comments may appear on a single line indented over to the tab setting of the code that follows.

```
if (argc > 1) {
        /* Get input file from command line. */
        if (freopen(argv[1], "r", stdin) == NULL)
                error("can't open %s\n", argv[1]);
}
```

Very short comments may appear on the same line as the code they describe, but should be tabbed over far enough to separate them from the statements. If more than one short comment appears in a block of code they should all be tabbed to the same tab setting.

```
if (a == 2)
        return(TRUE);           /* special case */
else
        return(isprime(a));/* works only for odd a */
```

6. Function Declarations

Each function should be preceded by a block comment prologue that gives the name and a short description of what the function does[20]. If the function returns a value, the type of the value returned should be alone on a line in column 1 (do not default to *int*). If the function does not return a value then it should not be given a return type. If the value returned requires a long explanation, it should be given in the prologue; otherwise it can be on the same line as the return type, tabbed over. The function name and formal parameters should be alone on a line beginning in column 1. Each parameter should be declared (do not default to *int*), with a comment on a single line. The opening brace of the function body should also be alone on a line beginning in column 1. The function name, argument declaration list, and opening brace should be separated by a blank line[21]. All local declarations and code within the function body should be tabbed over at least one tab.

If the function uses any external variables, these should have their own declarations in the function body using the *extern* keyword. If the external variable is an array the array bounds must be repeated in the *extern* declaration. There should also be *extern* declarations for all functions called by a given function. This is particularly beneficial to someone picking up code written by another. If a function returns a value of type other than *int*, it is required by the compiler that such functions be declared before they are used. Having the *extern* delcaration in the calling function's declarations section avoids all such problems[22].

---

19. A common practice in both Bell and local code is to use a space rather than a tab after the *. This is acceptable.

20. Discussion of non-trivial design decisions is also appropriate, but avoid duplicating information that is present in (and clear from) the code. It's too easy for such redundant information to get out of date.

21. Neither Bell nor local code has ever included these separating blank lines, and it is not clear that they add anything useful. Leave them out.

22. These rules tend to produce a lot of clutter. Both Bell and local practice frequently omits *extern* declarations for *static* variables and functions. This is permitted. Omission of declarations for standard library routines is also permissible, although if they *are* declared it is better to declare them within the functions that use them rather than globally.

In general each variable declaration should be on a separate line with a comment describing the role played by the variable in the function. If the variable is external or a parameter of type pointer which is changed by the function, that should be noted in the comment. All such comments for parameters and local variables should be tabbed so that they line up underneath each other. The declarations should be separated from the function's statements by a blank line.

A local variable should not be redeclared in nested blocks[23]. Even though this is valid C, the potential confusion is enough that *lint* will complain about it when given the **–h** option.

6.1. Examples

```
/*
 *	skyblue()
 *
 *	Determine if the sky is blue.
 */

int			/* TRUE or FALSE */
skyblue()

{
	extern int hour;

	if (hour < MORNING || hour > EVENING)
		return(FALSE);	/* black */
	else
		return(TRUE);	/* blue */
}

/*
 *	tail(nodep)
 *
 *	Find the last element in the linked list
 *	pointed to by nodep and return a pointer to it.
 */

NODE *			/* pointer to tail of list */
tail(nodep)

NODE *nodep;		/* pointer to head of list */

{
	register NODE *np;	/* current pointer advances to NULL */
	register NODE *lp;	/* last pointer follows np */

	np = lp = nodep;
	while ((np = np->next) != NULL)
		lp = np;
	return(lp);
}
```

---

23. In fact, avoid any local declarations that override declarations at higher levels.

7. Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces. The enclosed list should be tabbed over one more than the tab position of the compound statement itself. The opening left brace should be at the end of the line beginning the compound statement and the closing right brace should be alone on a line, tabbed under the beginning of the compound statement. Note that the left brace beginning a function body is the only occurrence of a left brace which is alone on a line.

7.1. Examples

```
if (expr) {
	statement;
	statement;
}

if (expr) {
	statement;
	statement;
} else {
	statement;
	statement;
}
```

Note that the right brace before the *else* and the right brace before the *while* of a *do-while* statement (below) are the only places where a right braces appears that is not alone on a line.

```
for (i = 0; i < MAX; i++) {
	statement;
	statement;
}

while (expr) {
	statement;
	statement;
}

do {
	statement;
	statement;
} while (expr);

switch (expr) {
case ABC:
case DEF:
	statement;
	break;
case XYZ:
	statement;
	break;
default:
	statement;
	break[24];
}
```

Note that when multiple *case* labels are used, they are placed on separate lines. The fall through feature of

24. This *break* is, strictly speaking, unnecessary, but it is required nonetheless because it prevents a fall-through error if another

the C *switch* statement should rarely if ever be used when code is executed before falling through to the next one. If this is done it must be commented for future maintenance.

```
if (strcmp(reply, "yes") == EQUAL) {
        statements for yes
        ...
} else if (strcmp(reply, "no") == EQUAL) {
        statements for no
        ...
} else if (strcmp(reply, "maybe") == EQUAL) {
        statements for maybe
        ...
} else {
        statements for none of the above
        ...
}
```

The last example is a generalized *switch* statement and the tabbing reflects the switch between exactly one of several alternatives rather than a nesting of statements.

8. Expressions

8.1. Operators

The old versions of equal-ops =+, =–, =*, etc. should not be used. The preferred use is +=, –=, *=, etc. All binary operators except . and –> should be separated from their operands by blanks[25]. In addition, keywords that are followed by expressions in parentheses should be separated from the left parenthesis by a blank[26]. Blanks should also appear after commas in argument lists to help separate the arguments visually. On the other hand, macros with arguments and function calls should not have a blank between the name and the left parenthesis. In particular, the C preprocessor requires the left parenthesis to be immediately after the macro name or else the argument list will not be recognized. Unary operators should not be separated from their single operand. Since C has some unexpected precedence rules, all expressions involving mixed operators should be fully parenthesized.

Examples

```
a += c + d;
a = (a + b) / (c * d);
strp->field = str.fl - ((x & MASK) >> DISP);
while (*d++ = *s++)
        ;       /* EMPTY BODY */
```

8.2. Naming Conventions

Individual projects will no doubt have their own naming conventions. There are some general rules however.

- An initial underscore should not be used for any user-created names[27]. UNIX uses it for names that the user should not have to know (like the standard I/O library)[28].

---

case is added later after the last one.

25. Some judgement is called for in the case of complex expressions, which may be clearer if the "inner" operators are not surrounded by spaces and the "outer" ones are.

26. *Sizeof* is an exception, see the discussion of function calls. Less logically, so is *return*.

27. Trailing underscores should be avoided too.

28. This convention is reserved for system purposes. If you must have your own private identifiers, begin them with a capital letter identifying the package to which they belong.

- Macro names, *typedef* names, and *define* names should be all in CAPS.
- Variable names, structure tag names, and function names should be in lower case[29]. Some macros (such as *getchar* and *putchar*) are in lower case since they may also exist as functions. Care is needed when interchanging macros and functions since functions pass their parameters by value whereas macros pass their arguments by name substitution[30].

8.3. Constants

Numerical constants should not be coded directly[31]. The *define* feature of the C preprocessor should be used to assign a meaningful name. This will also make it easier to administer large programs since the constant value can be changed uniformly by changing only the *define*. The enumeration data type is the preferred way to handle situations where a variable takes on only a discrete set of values, since additional type checking is available through *lint*.

There are some cases where the constants 0 and 1 may appear as themselves instead of as defines. For example if a *for* loop indexes through an array, then

```
for (i = 0; i < ARYBOUND; i++)
```

is reasonable while the code

```
fptr = fopen(filename, "r");
if (fptr == 0)
        error("can't open %s\n", filename);
```

is not. In the last example the defined constant *NULL* is available as part of the standard I/O library's header file *stdio.h* and must be used in place of the 0.

9. Portability

The advantages of portable code are well known. This section gives some guidelines for writing portable code, where the definition of portable is taken to mean that a source file contains portable code if it can be compiled and executed on different machines with the only source change being the inclusion of possibly different header files. The header files will contain defines and typedefs that may vary from machine to machine. Reference [1] contains useful information on both style and portability. Many of the recommendations in this document originated in [1]. The following is a list of pitfalls to be avoided and recommendations to be considered when designing portable code:

- First, one must recognize that some things are inherently non-portable. Examples are code to deal with particular hardware registers such as the program status word, and code that is designed to support a particular piece of hardware such as an assembler or I/O driver. Even in these cases there are many routines and data organizations that can be made machine independent. It is suggested that source file be organized so that the machine-independent code and the machine-dependent code are in separate files. Then if the program is to be moved to a new machine, it is a much easier task to determine what needs to be changed[32]. It is also possible that code in the machine-independent files may have uses in other programs as well.
- Pay attention to word sizes. The following sizes apply to basic types in C for the machines that will be used most at IH[33]:

---

29. It is best to avoid names that differ only in case, like *foo* and *FOO*. The potential for confusion is considerable.

30. This difference also means that carefree use of macros requires care when they are defined. Remember that complex expressions can be used as parameters, and operator-precedence problems can arise unless all occurrences of parameters in the definition have parentheses around them. There is little that can be done about the problems caused by side effects in parameters except to avoid side effects in expressions (a good idea anyway).

31. At the very least, any directly-coded numerical constant must have a comment explaining the derivation of the value.

32. If you *#ifdef* dependencies, make sure that if no machine is specified, the result is a syntax error, *not* a default machine!

33. The 3B is a Bell Labs machine. The VAX, not shown in the table, is similar to the 3B in these respects. The 68000 resembles either the pdp11 or the 3B, depending on the particular compiler.

type	pdp11	3B	IBM
char	8	8	8
short	16	16	16
int	16	32	32
long	32	32	32

In general if the word size is important, *short* or *long* should be used to get 16 or 32 bit items on any of the above machines[34]. If a simple loop counter is being used where either 16 or 32 bits will do, then use *int*, since it will get the most efficient (natural) unit for the current machine[35].

- Word size also affects shifts and masks. The code

  x &= 0177770

  will clear only the three rightmost bits of an *int* on a PDP11. On a 3B it will also clear the entire upper halfword. Use

  x &= ~07

  instead which works properly on all machines[36].

- Code that takes advantage of the two's complement representation of numbers on most machines should not be used. Optimizations that replace arithmetic operations with equivalent shifting operations are particularly suspect. You should weigh the time savings with the potential for obscure and difficult bugs when your code is moved, say, from a 3B to a 1A.

- Watch out for signed characters. On the PDP-11, characters are sign extended when used in expressions, which is not the case on any other machine. In particular, *getchar* is an integer-valued function (or macro) since the value of *EOF* for the standard I/O library is −1, which is not possible for a character on the 3B or IBM[37].

- The PDP-11 is unique among processors on which C exists in that the bytes are numbered from right to left within a word. All other machines (3B, IBM, Interdata 8/32, Honeywell) number the bytes from left to right[38]. Hence any code that depends on the left-right orientation of bits in a word deserves special scrutiny. Bit fields within structure members will only be portable so long as two separate fields are never concatenated and treated as a unit[39]. [1,3]

- Do not default the boolean test for non-zero, i.e.

  if (f() != FAIL)

  is better than

  if (f())

---

34. Any unsigned type other than plain *unsigned int* should be *typedef*ed, as such types are highly compiler-dependent. This is also true of long and short types other than *long int* and *short int*. Large programs should have a central header file which supplies *typedefs* for commonly-used width-sensitive types, to make it easier to change them and to aid in finding width-sensitive code.

35. Beware of making assumptions about the size of pointers. They are not always the same size as *int*. Nor are all pointers always the same size, or freely interconvertible. Pointer-to-character is a particular trouble spot on machines which do not address to the byte.

36. The or operator (|) does not have these problems, nor do bitfields (which, unfortunately, are not very portable due to defective compilers).

37. Actually, this is not quite the real reason why *getchar* returns *int*, but the comment is valid: code which assumes either that characters are signed or that they are unsigned is unportable. It is best to completely avoid using *char* to hold numbers. Manipulation of characters as if they were numbers is also often unportable.

38. Actually, there are some more right-to-left machines now, but the comments still apply.

39. The same applies to variables in general. Alignment considerations and loader peculiarities make it very rash to assume that two consecutively-declared variables are together in memory, or that a variable of one type is aligned appropriately to be used as another type.

even though *FAIL* may have the value 0 which is considered to mean false by C[40]. This will help you out later when somebody decides that a failure return should be −1 instead of 0[41].

- Be suspicious of numeric values appearing in the code. Even simple values like 0 or 1 could be better expressed using defines like *FALSE* and *TRUE* (see previous item)[42]. Any other constants appearing in a program would be better expressed as a defined constant. This makes it easier to change and also easier to read.
- Become familiar with existing library functions and defines[43]. You should not be writing your own string compare routine, or making your own defines for system structures[44]. Not only does this waste your time, but it prevents your program from taking advantage of any microcode assists or other means of improving performance of system routines[45].
- Use *lint*. It is a valuable tool for finding machine-dependent constructs as well as other inconsistencies or program bugs that pass the compiler[46].

10. Lint

Lint is a C program checker [2] that examines C source files to detect and report type incompatibilities, inconsistencies between function definitions and calls, potential program bugs, etc. It is expected that projects will require programs to use *lint* as part of the official acceptance procedure[47]. In addition, work is going on in department 5521 to modify *lint* so that it will check for adherence to the standards in this document.

It is still too early to say exactly which of the standards given here will be checked by *lint*. In some cases such as whether a comment is misleading or incorrect there is little hope of mechanical checking. In other cases such as checking that the opening brace of a function body is alone on a line in column 1, the test has already been added[48]. Future bulletins will be used to announce new additions to *lint* as they

---

40. A particularly notorious case is using *strcmp* to test for string equality, where the result should *never ever* be defaulted. The preferred approach is to define a macro *STREQ*:

```
#define STREQ(a, b) (strcmp((a), (b)) == 0)
```

41. An exception is commonly made for predicates, which are functions which meet the following restrictions:

- Has no other purpose than to return true or false.
- Returns 0 for false, 1 for true, nothing else.
- Is named so that the meaning of (say) a 'true' return is absolutely obvious. Call a predicate *isvalid* or *valid*, not *checkvalid*.

42. Actually, *YES* and *NO* often read better.
43. But not *too* familiar. The internal details of library facilities, as opposed to their external interfaces, are subject to change without warning. They are also often quite unportable.
44. Or, especially, writing your own code to control terminals. Use the *termcap* package.
45. It also makes your code less readable, because the reader has to figure out whether you're doing something special in that reimplemented stuff to justify its existence. Furthermore, it's a fruitful source of bugs.
46. The use of *lint* on all programs is strongly recommended. It is difficult to eliminate complaints about functions whose return value is not used (in the current version of C, at least), but most other messages from *lint* really do indicate something wrong. The −h, −p, −a, −x, and −c options are worth learning. All of them will complain about some legitimate things, but they will also pick up many botches. Note that −p checks function-call type-consistency for only a subset of Unix library routines, so programs should be linted both with and without this option for best "coverage".
47. Yes.
48. Little of this is relevant at U of T. The version of *lint* that we have lacks these mods.

occur.

It should be noted that the best way to use *lint* is not as a barrier that must be overcome before official acceptance of a program, but rather as a tool to use whenever major changes or additions to the code have been made. *Lint* can find obscure bugs and insure portability before problems occur.

11. Special Considerations

This section contains some miscellaneous do's and don'ts.

- Don't change syntax via macro substitution. It makes the program unintelligible to all but the perpetrator.
- There is a time and a place for embedded assignment statements[49]. In some constructs there is no better way to accomplish the results without making the code bulkier and less readable. The *while* loop in section 8.1 is one example of an appropriate place. Another is the common code segment:

```
while ((c = getchar()) != EOF) {
        process the character
}
```

  Using embedded assignment statements to improve run-time performance is also possible. However, one should consider the tradeoff between increased speed and decreased maintainability that results when embedded assignments are used in artificial places. For example, the code:

```
a = b + c;
d = a + r;
```

  should not be replaced by

```
d = (a = b + c) + r;
```

  even though the latter may save one cycle. Note that in the long run the time difference between the two will decrease as the optimizer gains maturity, while the difference in ease of maintenance will increase as the human memory of what's going on in the latter piece of code begins to fade[50].
- There is also a time and place for the ternary *? :* operator and the binary comma operator. The logical expression operand before the *? :* should be parenthesized:

```
(x >= 0) ? x : –x
```

  Nested *? :* operators can be confusing and should be avoided if possible. There are some macros like *getchar* where they can be useful. The comma operator can also be useful in *for* statements to provide multiple initializations or incrementations.
- Goto statements should be used sparingly as in any well-structured code[51]. The main place where they can be usefully employed is to break out of several levels of *switch*, *for*, and *while* nesting[52], e.g.

---

49. The ++ and — operators count as assignment statements. So, for many purposes, do functions with side effects.

50. Note also that side effects within expressions can result in code whose semantics are compiler-dependent, since C's order of evaluation is explicitly undefined in most places. Compilers do differ.

51. The *continue* statement is almost as bad. *Break* is less troublesome.

52. The need to do such a thing may indicate that the inner constructs should be broken out into a separate function, with a success/failure return code.

```
    for (...)
        for (...) {
            ...
            if (disaster)
                goto error;
        }
    ...
error:
    clean up the mess
```

When a *goto* is necessary the accompanying label should be alone on a line and tabbed one tab position to the left of the associated code that follows.

- This committee recommends that programmers not rely on automatic beautifiers for the following reasons. First, the main person who benefits from good program style is the programmer himself. This is especially true in the early design of handwritten algorithms or pseudo-code. Automatic beautifiers can only be applied to complete, syntactically correct programs and hence are not available when the need for attention to white space and indentation is greatest. It is also felt that programmers can do a better job of making clear the complete visual layout of a function or file, with the normal attention to detail of a careful programmer[53]. Sloppy programmers should learn to be careful programmers instead of relying on a beautifier to make their code readable. Finally, it is felt that since beautifiers are non-trivial programs that must parse the source, the burden of maintaining them in the face of the continuing evolution of C is not worth the benefits gained by such a program.

12. Project Dependent Standards

Individual projects may wish to establish additional standards beyond those given here. The following issues are some of those that should be adddressed by each project program administration group.

- What additional naming conventions should be followed? In particular, systematic prefix conventions for functional grouping of global data and also for structure or union member names can be useful.
- What kind of include file organization is appropriate for the project's particular data hierarchy?
- What procedures should be established for reviewing *lint* complaints? A tolerance level needs to be established in concert with the *lint* options to prevent unimportant complaints from hiding complaints about real bugs or inconsistencies.
- If a project establishes its own archive libraries, it should plan on supplying a lint library file [2] to the system administrators. This will allow *lint* to check for compatible use of library functions.

13. Conclusion

A set of standards has been presented for C programming style. One of the most important points is the proper use of white space and comments so that the structure of the program is evident from the layout of the code. Another good idea to keep in mind when writing code is that it is likely that you or someone else will be asked to modify it or make it run on a different machine sometime in the future.

As with any standard, it must be followed if it is to be useful. The Indian Hill version of *lint* will enforce those standards that are amenable to automatic checking. If you have trouble following any of these standards don't just ignore them. Programmers at Indian Hill should bring their problems to the Software Development System Group (Lee Kirchhoff, contact) in department 5522. Programmers outside Indian Hill should contact the Processor Application Group (Layne Cannon, contact) in department 5512[54].

---

53. In other words, some of the visual layout is dictated by intent rather than syntax. Beautifiers cannot read minds.

54. At U of T Zoology, it's Henry Spencer in 336B.

References

[1] B.A. Tague, "C Language Portability", Sept 22, 1977. This document issued by department 8234 contains three memos by R.C. Haight, A.L. Glasser, and T.L. Lyon dealing with style and portability.

[2] S.C. Johnson, "Lint, a C Program Checker", Technical Memorandum, 77-1273-14, September 16, 1977.

[3] R.W. Mitze, "The 3B/PDP-11 Swabbing Problem", Memorandum for File, 1273-770907.01MF, September 14, 1977.

[4] R.A. Elliott and D.C. Pfeffer, "3B Processor Common Diagnostic Standards- Version 1", Memorandum for File, 5514-780330.01MF, March 30, 1978.

[5] R.W. Mitze, "An Overview of C Compilation of UNIX User Processes on the 3B", Memorandum for File, 5521-780329.02MF, March 29, 1978.

[6] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall 1978.

```
/*
 * The C Style Summary Sheet                                              Block comment,
 * by Henry Spencer, U of T Zoology                          describes file.
 */

#include <errno.h>                                                        Headers; don't nest.

typedef int         SEQNO;          /* ... */                             Global definitions.
#define             STREQ(a, b)     (strcmp((a), (b)) == 0)

static char *foo = NULL;            /* ... */                Global declarations.
struct bar {                                                                          Static when
                    SEQNO alpha;    /* ... */
#                                   define       NOSEQNO     0
                    int beta;       /* ... */                             Don't assume 16 bits.
};

/*
 * Many unnecessary braces, to show where.                   Functions.
 */
static int                          /* what is returned */                Don't default int.
bletch(a)
int a;                                           /* ... */                            Don't defau
{
                    int bar;                     /* ... */
                    extern int errno;   /* ..., changed here */
                    extern char *index();

                    if (foobar() != FAIL) {                               if (!isvalid()) {
                                    return(OK);                                       errno = ER
                    }                                                                 } else {

                    while (x == (y & MASK)) {                }
                                    f += (x >= 0) ? x : −x;
                    }                                                                 for (i = 0; i

                    do {                                                              }
                                    /* Avoid nesting ?: */
                    } while (index(a, b) != NULL);           if (STREQ(x, "foo")) {

                    switch (...) {                                        } else if (STREQ(x, "bar")) {
                    case ABC:                                                         x &= ˜077;
                    case DEF:                                             } else if (STREQ(x, "ugh")) {
                                    printf("...", a, b);                  /* Avoid gotos */
                                    break;                                            } else {
                    case XYZ:                                                         /* and conti
                                    x = y;                                            }
                                    /* FALLTHROUGH */
                    default:                                              while ((c = getc()) != EOF)
                                    /* Limit imbedded =s. */              ;           /* NULLB(
                                    break;
                    }
}
```
